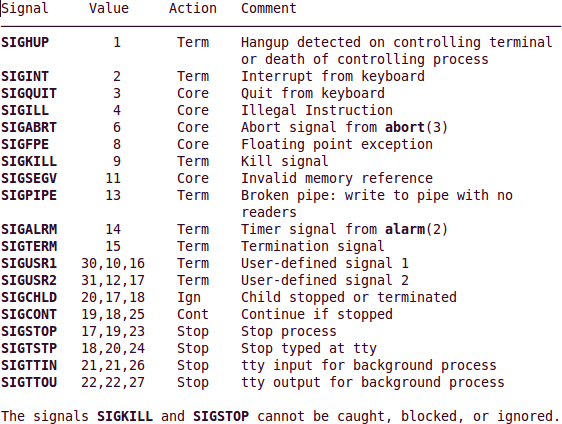
Giao tiếp giữa các tiến trình

# Bài tập 1 :

Một tiến trình có thể gửi tới một tiến trình khác một Signal . Các Signal phần lớn được gửi đi từ kernel khi một sự kiện xảy ra, ví dụ : xuất hiện một ngoại lệ hardware, xuất hiện một sự kiện software, người dùng nhấn phím đặc biệt … Các tiến trình cũng có thể dùng Signal như một cách để giao tiếp với nhau.

Khi một sự kiện phát sinh, một Signal tương ứng được gửi đi. Tiến trình nhận Signal sẽ thực hiện hành động đáp ứng nhất định (default). Mỗi Signal được gán một số nguyên duy nhất và một tên (symbolic name). Bảng sau liệt kê một số Signal trong Linux .(*tham khảo man 7 signal*)



Trong Linux**, kill là một lệnh** được dùng để gửi signal tới các tiến trình:

* Xem danh sách các Signal :

## kill -l

**kill -l SIGTERM**

* Gửi signal SIGTERM để kết thúc một tiến trình đang chạy :

// tạo một tiến trình đang chạy

## nano

// gửi signal TERM đế kết thúc tiến trình (\*)

## kill -SIGTERM <PID\_ls>

// xem thông tin về tiến trình

## ps –a

* Đọc man kill , cho biết gọi kill bằng cách khác tương đương với (\*) ?
* Thực hiện tương tự để kết thúc một tiến trình bằng signal **SIGKILL** ? Phân biệt 2 signal **SIGTERM và SIGKILL** ?
* Thực hiện sử dụng 2 signal : **SIGSTOP , SIGCONT** để tạm ngưng rồi resume tiến trình ?
* Tìm hiểu lệnh trong Windows tương tự kill ?

# Bài tập 2 :

Kill() system call được sử dụng để một tiến trình gửi một signal tới tiến trình khác.

Khi một tiến trình nhận được một signal , thông thường nó thực hiện đáp ứng theo default. Ta có thể hàm signal() để thay đổi hành vi đáp ứng của tiến trình.

**int kill(pid\_t pid, int sig);**

* + Pid : id của tiến trình nhận signal . Pid cũng có thể mang một số giá trị đặc biệt.
  + Sig : signal gửi cho tiến trình ( dạng số )
  + Returns 0 on success, or –1 on error

*( tham khảo man 2 kill )*

**sighandler\_t signal(int signum, sighandler\_t handler);**

* signum: là số hiệu signal mà ta muốn thay đổi hành vi (trừ SIGKILL hay SIGSTOP) - dạng số hay symbolic
* handler: hành vi mới đối với signal, các giá trị có thể là:
  + SIG\_DFL: thiết lập lại hành vi về mặc định (default)
  + SIG\_IGN: lờ đi (ignore) signal tương ứng
  + Tham chiếu đến hàm xử lý sự kiện mới (signal-handler) do người dùng tự định nghĩa
* Returns previous signal disposition on success, or SIG\_ERR on error

*( tham khảo man signal )*

1. Hai chương trình sau đây sử dụng **signal()** chờ nhận signal SIGINT (phím ctrl-C). Chương trình thứ nhất (vd1) : thực thi catch khi nhận signal. Chương trình thứ hai (vd2) bỏ qua signal này.

## Đọc hiểu chương trình, thực thi và xem kết quả ?

/\*vd1 : dap ung bang ham catch \*/

*#include <stdio.h> #include <signal.h> #include <unistd.h> #include <stdlib.h>* **void catch(int sig)** {

printf(" I received signal %d \n ", sig );

}

**int main()** {

if (signal(SIGINT, catch) == SIG\_ERR) {

perror("SIGINT\n"); exit(3);

}

while (1)

sleep(1); return 0;

}

/\*vd2 : bo qua SIGINT \*/ *#include <stdio.h> #include <signal.h> #include <unistd.h> #include <stdlib.h>*

**int main()** {

if (signal(SIGINT, SIG\_IGN) == SIG\_ERR)

perror("SIGINT\n"); exit(3);

}

while (1)

sleep(1); return 0;

}

{

1. **Đọc hiểu chương trình sau , thực thi và xem kết quả ?**

/\* *In alarm.c, the first function, ding, simulates an alarm clock*.\*/ #include <signal.h>

#include <stdio.h> #include <unistd.h> #include <stdlib.h>

static int alarm\_fired = 0;

**void ding(int sig)** */\*Func for SIGALARM\*/*

{

alarm\_fired = 1;

}

*/\* In main, we tell the child process to wait for five seconds before sending a SIGALRM signal to its parent. \*/*

**int main()**

{

pid\_t pid;

printf("alarm application starting\n"); pid = fork();

switch(pid) { case -1:

perror("fork failed"); exit(1);

case 0: sleep(5);

kill(getppid(), SIGALRM); */\*Get signal & send it to parent\*/*

exit(0);

}

*/\* The parent process arranges to catch SIGALRM with a call to signal and then waits for the inevitable. \*/*

printf("waiting for alarm to go off\n"); signal(SIGALRM, ding);

pause();

if (alarm\_fired) printf("Ding!\n");

printf("done\n"); exit(0);

}

*/\* end \*/*

# Bài tập 3:

1. Viết chương trình nhận các signal ( SIGINT, SIGTERM, SIGKILL…).

Khi nhận được signal, tiến trình hiển thi thông báo đã nhận được signal xx .

1. Khi tiến trình con thay đổi trạng thái - terminate, stop hay resume , kernel sẽ gửi tới tiến trình cha signal SIGCHLD . Hãy viết chương trình để tiến trình cha khi nhận được SIGCHLD thì thực thi wait() , tránh tình trạng zombie process.

# Bài tập 4:

1. Các lời gọi hệ thống khi thực hiện nhập/xuất đều tham chiếu đến các “file” đang mở thông qua *file descriptor* (một số nguyên dương). “File” có thể là pipes, FIFOs, sockets, terminals, devices, và regular files.

Chương trình shell, sử dụng terminal để nhập/xuất dữ liệu, thường tham chiếu tới 3 *file descriptor* : 0 (standard input), 1 (standard output) và 2 (standard error). Khi một chương trình được gọi từ shell , nó cũng có các *file descriptor* này.

Trong Linux, có thể xem tập các *file descriptor* của 1 tiến trình bằng cách đọc :

/proc/<PID>/fd .

**Thực hiện xem các *file descriptor* của 1 tiến trình ?**

* Trên terminal thứ nhất, thực thi lệnh : **more /var/log/syslog**
* Trên terminal thứ hai:
  + xem pid của tiến trình more
  + xem các *file descriptor* của tiến trình này : **ls -l /proc/<PID>/fd**

1. Các tiến trình có thể chia sẽ dữ liệu với nhau bằng cách sử dụng pipe. pipe là một kênh truyền dòng bytes 1 chiều giữa các tiến trình. Thông qua pipe, một tiến trình xuất dữ liệu vào pipe, và một tiến trình đọc dữ liệu từ pipe. Các Linux shell đều hỗ trợ pipe thông qua toán tử |

## ls / | sort

**cat /etc/passwd | more**

# Bài tập 5:

Hàm pipe() tạo một pipe , đồng thời tạo 2 *file descriptor* mới cho tiến trình : một *file descriptor* gắn với đầu đọc của pipe, và một *file descriptor* gắn với đầu ghi của pipe.

**#include<unistd.h>**

**int pipe(int filedes[2]);**

***//*** *filedes[0] gắn với đầu đọc của pipe và filedes[1] gắn với đầu ghi của pipe*

Có thể sử dụng read() và write() system calls để thực hiện đọc/ghi vào pipe.

Pipe() được dùng để giao tiếp giữa tiến trình cha và tiến trình con. Lời gọi pipe() sẽ đặt ngay trên fork(). Khi tiến trình con được tạo bởi fork(), nó sẽ có các *file descriptor* giống hệt các *file descriptor* của tiến trình cha. Lúc này, cả 2 tiến trình cha và con đều có các *file descriptor* tham chiếu tới cùng một pipe.

Close() được dùng để đóng một *file descriptor* – *file descriptor* không còn tham chiếu tới bất kz pipe nào và không thể sử dụng lại. Một tiến trình chỉ ghi , sẽ dùng close() đóng read descriptor trước khi ghi vào pipe. Tương tự, một tiến trình chỉ đọc, sẽ dùng close() đóng write descriptor trước khi đọc pipe. Khi pipe không còn có bất kz *file descriptor* nào tham chiếu tới, pipe sẽ được hủy.

*(Tham khảo man 7 pipe, man 2 pipe, man read, man 2 write, man close)*

**Chương trình sau minh họa sử dụng pipe để giao tiếp giữa tiến trình cha và tiến trình con. Đọc hiểu, thực thi và xem kết quả của chương trình ?**

/\*giao tiếp giữa tiến trình cha và tiến trình con \*/ #include <stdio.h>

#include <stdlib.h> #include <unistd.h>

**int main(int argc, char \*argv[])**

{

int fd[2]; int val = 0;

pipe(fd);

// create pipe descriptors

if (fork() != 0)

{

// parent: writing only, so close read-descriptor. close(fd[0]);

// send the value on the write-descriptor. val = 100;

write(fd[1], &val, sizeof(val));

printf("Parent(%d) send value: %d\n", getpid(), val);

// close the write descriptor close(fd[1]);

wait(NULL);

}

else

{ // child: reading only, so close the write-descriptor close(fd[1]);

// now read the data (will block) read(fd[0], &val, sizeof(val));

printf("Child(%d) received value: %d\n", getpid(), val);

// close the read-descriptor close(fd[0]);

}

exit(0);

}

# Bài tập 6 : Sử dụng pipe để giao tiếp giữa process cha và nhiều process con .

## Đọc hiểu, thực thi và xem kết quả của chương trình sau ?

